Prüfung

Prof. Dr.-Ing. K. D. Müller-Glaser

Informationstechnik

WS 2008/09

Institut für Technik der Informationsverarbeitung, Universität Karlsruhe

Musterlösung zur Klausur WS 2008/09

Do., 5.3.2009

Hinweise zur Klausur

Hilfsmittel

Zur Prüfung sind keine Hilfsmittel zugelassen. Nicht erlaubt sind insbesondere die Verwendung eines Mobiltelefons und jegliche Kommunikation mit anderen Personen.

Prüfungsdauer

Die Prüfungsdauer beträgt 120 Minuten.

Prüfungsunterlagen

Die Prüfungsunterlagen bestehen aus insgesamt 23 Seiten Aufgabenblättern (einschließlich diesem Titelblatt und zusätzlicher Lösungsblätter).

Bitte vermerken Sie vor der Bearbeitung der Aufgaben auf jeder Seite oben Ihren Namen, auf der ersten Seite zusätzlich die Matrikelnummer!

Falls Sie zusätzliche Blätter zur Lösung der Aufgaben benötigen, verwenden Sie die zusätzlichen Seiten am Ende der Klausur bzw. fragen Sie nach zusätzlichem Papier. Auf jedes zusätzliche Lösungsblatt ist neben dem Namen auch die Aufgaben- und die Seitennummer mit einzutragen. Vermeiden Sie das Beschreiben der Rückseiten. Die Verwendung von eigenen Blättern ist nicht erlaubt. Am Ende der Prüfung sind die 23 Seiten Aufgaben- und Lösungsblätter und alle verwendeten zusätzlichen Lösungsblätter abzugeben.
Verwenden Sie zum Bearbeiten der Aufgaben nur dokumentenechte Schreibgeräte – keinen Bleistift, keine Rotstifte!

Wie verabredet enthält die Klausur mehr Aufgaben als, bei einer richtigen Lösung, zum Erreichen einer sehr guten Note (1,0) notwendig sind. (Nutzen Sie die Auswahlmöglichkeit)
Die mit * gekennzeichneten Teilaufgaben können Sie nicht unabhängig von den anderen Teilaufgaben lösen.

Aufgabe	1	2	3	4	5	6	7	8	Σ
≈ Gewichtung ca. [%]	13	13	11	10	14	15	11	13	100
Ergebnis [P]									

Aufgabe 1 Allgemeine Fragen

Beantworten Sie folgende Fragen:

A) Nennen Sie den Unterschied zwischen Funktionen einer imperativen Programmiersprache und Methoden einer objektorientierten Programmiersprache.

Funktionen sind unabhängige Unterprogramme. Der Programmierer muss sicherstellen dass die Eingangsdaten im richtigen Format sind.

Methoden sind an ein Objekt gekoppelt und arbeiten mit dessen Daten.

B) Erklären Sie den Unterschied zwischen einer Referenz und einem Zeiger.

Eine Referenz stellt einen anderen Namen für ein bestimmtes, existierendes Objekt dar und benötigt keinen Speicherplatz.

Ein Zeiger ist eine eigene Variable die die Adresse eines anderen Objekts enthält. Ein Zeiger läßt sich ändern d.h. auf ein anderes Objekt verweisen.

C) Welche Eigenschaften muss eine Berechnungsvorschrift erfüllen, damit sie formal ein Algorithmus ist? Nennen Sie 4 Begriffe und erklären Sie sie kurz.

- **Finitheit:** das Verfahren muss in einem endlichen Text eindeutig beschreibbar sein.
- **Ausführbarkeit:** jeder Schritt des Verfahrens muss tatsächlich ausführbar sein
- **Dynamische Finitheit** (Platzkomplexität): das Verfahren darf zu jedem Zeitpunkt nur endlich viel Speicherplatz benötigen
- **Terminierung** (Zeitkomplexität): das Verfahren darf nur endlich viele Schritte benötigen

Halbe Punkte gibt es für:

- **Determiniertheit:** der Algorithmus muss bei denselben Voraussetzungen das gleiche Ergebnis liefern
- **Determinismus:** die nächste anzuwendende Regel im Verfahren ist zu jedem Zeitpunkt eindeutig definiert

D)

Richtig	*Falsch*	
☒ X	☐	Jeder iterative Algorithmus läßt sich auch als rekursiver Algorithmus darstellen.
☒ X	☐	Insertion Sort ist ein stabiles Sortierverfahren.
☐	☒ X	Insertion Sort hat im Optimalfall eine Komplexität in O(log n)
☐	☒ X	Im Gegensatz zu einem Array eignet sich eine Liste besonders gut für die binäre Suche.

E) Wann heißen zwei Graphen G1 und G2 isomorph?

Zwei Graphen G1 und G2 heißen isomorph, wenn es eine bijektive Abbildung der Knoten- und Kantenmengen von G1 auf die entsprechenden Mengen von G2 gibt, so dass die Inzidenzbeziehungen erhalten bleiben.

F) Welche zusätzlichen Eigenschaften muss ein Graph erfüllen, damit er ein Baum ist?

Ein Baum ist zyklenfrei und zusammenhängend.

G) Nennen Sie 4 Merkmale guter Software aus Qualitätssicht.

Zuverlässigkeit
Wartbarkeit
Effizienz
Benutzerfreundlichkeit
Nützlichkeit

Aufgabe 2 C++ Verständnisfragen

Beantworten Sie folgende Fragen:

A) Welche Ausgabe erzeugt folgende C++ Zeile?

```
cout << "Alles \\\neu!" << endl;
```

```
Alles \
eu!
```

B) Gegeben ist der folgende kurze Programmausschnitt. Nehmen Sie an, dass das Programm korrekt und fehlerfrei kompiliert worden ist. Welches sind die Ausgaben auf dem Bildschirm aufgrund der dargestellten Zeilen C++ Programmcode?

```
int main() {
      int i = 5;

      cout << 5/2 << endl;        2
      cout << 5.0/2 << endl;      2.5
      cout << i++ << endl;        5
      cout << ++i << endl;        7
}
```

C) Welche Werte enthalten die Variablen s1, s2 und s3 nach dem Aufruf des folgenden C++ Codes ?

```
string s1;
string s2(s1);
string s3("s2");
```

s1 und s2 sind leer
s3 enthält den string "s2"

D) Nach Ausführung der folgenden Anweisungen, welchen Wert speichert der String *s*?

```
string s("IN EZ");
s += "!";
int pos = s.find(" ");
s.insert(pos+1, "FO RUL");
s.erase(pos, 1);
```

s = `"INFO RULEZ!"`

E) Nach Ausführung der folgenden Schleife, welchen Wert speichert die Variable *x*?

```
int x=0;
for (int i=0; i<4; x += i++);
```

x = 6

F) Was ist das Ergebnis des folgenden C++ Ausdrucks? Dabei ist die Variable *Z* vom Typ *bool*.

a) `Z = !((10 < 20) ^ (5 % 3))` ☐ *true* ☒ *false*

b) `Z = ((5 < 10) - (3 & 1)) == 0;` ☒ *true* ☐ *false*

G) Finden Sie den Fehler im folgenden C++ Programmausschnitt. Beschreiben und korrigieren Sie diesen Fehler.

```
const int array_size = 10;
int Arr[array_size];
for (int i = 1; i <= array_size; ++i)
    Arr[i] = i;
```

→ Fehler: Array-Grenzen wurden nicht eingehalten
→ Korrigieren: entweder i < array_size
oder Arr[i-1]=i oder Arr[i-1] = i-1

H) Was berechnet die Funktion *Funktion( )* ?

```
#include <iostream>

int Funktion (int val)
{
    if (val > 1)
       return Funktion(val-1) * val;
    else
       return 1;
}
```

Antwort: Funktion berechnet die Fakultät der Zahl val

I) Welchen Wert enthält die Variable *c* nach Ausführung der *switch*-Anweisung?

```
int i = 5;
char c;

switch (i)
{
   default : c = 'X';
   case 1: c = 'T'; break;
   case 2: c = 'B';
}
```

Beachten: die break Anweisung fehlt

c = **'T'**

J) Finden Sie jeweils einen Fehler in den zwei folgenden Quellcodeabschnitten?

a)
```
class A
{
  private:
       string typ;
  public:
       void set(string s) const {
            typ = s;
       }
       string get() const {
            return typ;
       }
};
```

Antwort: Die Funktion set darf nicht konstant sein, um auf typ schreiben zu dürfen.

b)
```
class B
{
  private:
       double Arr[5];
  public:
       double feld(int i) {
            return Arr[i];
       };
};
```

Antwort: Hinter der geschweiften Klammer der Funktion feld ist ein Semikolon zu viel.

Aufgabe 3 Polynom- und Matrix-Berechnung

Das folgende C++ Programm soll ein einzugebendes Polynom an einer ebenfalls einzugebenden Stelle *x* auswerten. Der Grad des Polynoms wird dabei bestimmt durch die Variable *grad*. Das Polynom hat dabei die allgemeine Form:

$$y = k_0 + k_1 \cdot x^1 + \ldots + k_n \cdot x^n ; n < 100$$

Leider wurde festgestellt, dass das Programm nicht korrekt arbeitet und nicht vollständig ist. Die Funktion *potenz()* soll im Programm die Potenz des Wertes *x* mit dem Exponenten *i* berechnen (x^i) und das Ergebnis als Rückgabewert zurückgeben.

➔ Aufgaben siehe nächste Seiten!!!

```
01 #include <iostream>
02 using namespace std;
03
04 double potenz(double x, int i)
05 {
06
07
08
09
10
11 }
12
13 int main()
14 {
15   int grad;
16   double k[100];
17   double x, erg_y = 0.0;
18
19   cout << "Grad des Polynoms eingeben: ";
20   cin >> grad;
21
22   if ((grad >= 0) && (grad < 100))
23   {
24     for (int i = 0; i <= grad; i++)
25       cout << "Koeffizient von x hoch " << i << " eingeben: ";
26       cin >> k[i];
27     cout << "Auszuwertendes x eingeben: x = ";
28     cin >> x;
29
30     for (int i = 0; i <= grad; i++)
31       erg_y = k[i] * potenz(x, i);
32     cout << "Wert an der Stelle x = " << x;
33     cout << ": " << erg_y << endl;
34   }
35   else
36     cout << "Fehler: Es muss gelten: 0 <= Grad < 100" << endl;
37   return 0;
38 }
```

A) In der Funktion *main()* befinden sich 2 semantische Fehler. Nennen Sie die entsprechen Zeilen, beschreiben Sie die Fehler und korrigieren Sie diese.

Zeile 24 & 26: Fehlende Klammern für die for-Schleife
Zeile 31: um den Wert an der Stelle x zu berechnen, müssen die einzelnen Polynome addiert werden. Daher müsste Zeile 31 folgendermaßen aussehen:

```
erg_y += k[i] * potenz(x, i);
```

B) Ergänzen Sie die C++ Funktion *potenz()*. Diese soll die Potenz des Wertes *x* mit dem Exponenten *i* berechnen (x^i) und das Ergebnis als Rückgabewert zurückgeben.

```
double potenz(double x, int i)
{
 double hilfe = 1;

 for (int j = 1; j <= i; j++)
     hilfe = hilfe * x;
 return hilfe;









}
```

C) Als Erweiterung soll das Programm nun so verändert werden, dass anstatt eines Arrays von 100 Werten ein dynamisches Array verwendet wird, sodass nur soviel Speicherplatz wie benötigt reserviert wird. Schreiben Sie eine Zeile C++ Programmcode, welche ein dynamisches Array vom Typ *double* mit dem Namen *k2* anlegt. Die Größe des Arrays ist enthalten in der Integer-Variablen *grad*. (Diese Zeile würde in Zeile 21 des Programms eingefügt werden). Das restliche Programm müssen Sie nicht verändern.

```
double* k2 = new double[grad];
```

D) Wie kann man den reservierten Speicher für das in Teilaufgabe C) angelegte dynamische Array mit einer Zeile C++ Programmcode wieder freigeben?

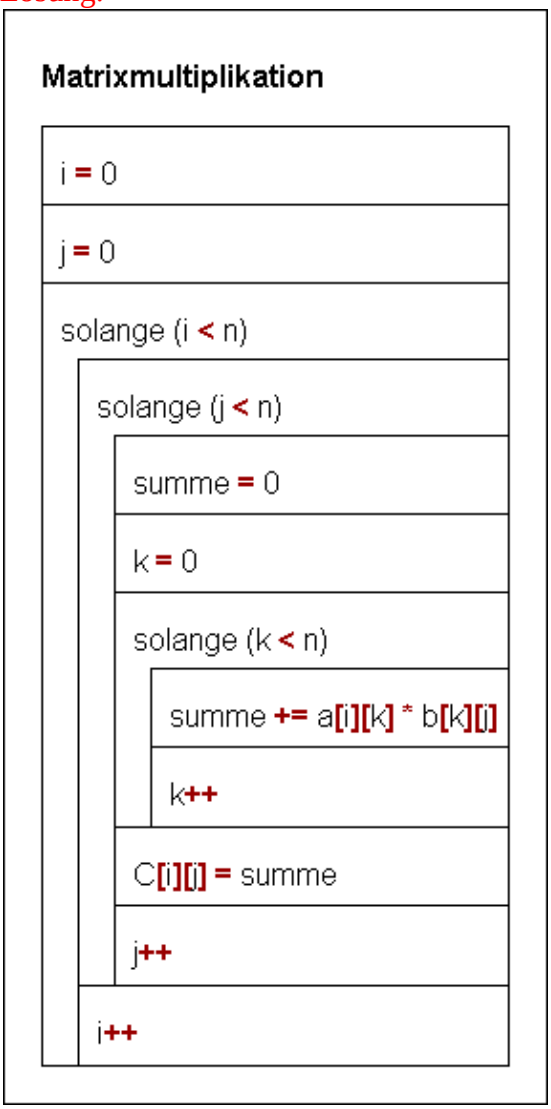
```
delete[] k2;
```

E) Das oben beschriebene Mathematikprogramm soll um einen Algorithmus zur Matrizenmultiplikation von zwei (n*n) Matrizen ergänzt werden. Beschreiben Sie diesen Algorithmus mit Hilfe eines Nassi-Shneiderman Diagramms. Dabei sind die zwei Matrizen in den zweidimensionalen Arrays *A[n][n]* und *B[n][n]* und die Größe in der Konstanten *n* gespeichert. Das Ergebnis soll am Ende in einem zweidimensionalen Array *C[n][n]* gespeichert werden.

Für die Indizes der (n*n) Matrizen gilt dabei:

$A = \quad a_{ij} \quad _{i=1\ldots n, j=1\ldots n}$ und $B = \quad b_{ij} \quad _{i=1\ldots n, j=1\ldots n}$

Lösung:



Aufgabe 4 Variablenübergabe

Gegeben sei ein Ausschnitt aus einem C++ Programm:

```
01 #include <iostream>
02
03 int i;
04 int b[2];
05
06 void p(int x)
07 {
08                  // -3-
09     i = i + 1;
10     x = x + 2;
11                  // -4-
12 }
13
14 int main()
15 {
16     b[0] = 10;
17     b[1] = 20;
18     i = 0;
19                 // -1-
20     p (b[i]);
21                 // -2-
22     return 0;
23 }
```

A) Tragen Sie in die unten stehenden Tabellen die Werte ein, die die angegebenen Variablen vor der Abarbeitung der jeweiligen kommentierten Programmzeile haben.

Hauptprogramm

Stelle	b[0]	b[1]	i
-1-	10	20	0
-2-	10	20	1

Unterprogramm

Stelle	x
-3-	10
-4-	12

B) Wie nennt man die im Programm angewendete Variablenübergabe an die Funktion *p()*?

Call by Value / Wertübergabe

C) *Nennen Sie die zwei anderen Arten (unterschiedlich zu B) der Variablenübergabe?

Call by Referenz & Call by Zeigervariable / Referenzübergabe & Zeigerübergabe

D) *Verändern Sie das Programm entsprechend den unter C) genannten Arten der Variablenübergabe und schreiben Sie die veränderten Zeilen (mit Zeilennummer) für jede der zwei anderen Übergabearten auf. Verändern Sie dabei die Variablenübergabe und das restliche Programm (wenn nötig), sodass es fehlerfrei kompiliert werden kann und lauffähig ist. Die Werte der Variablen werden sich hierdurch verändern, füllen Sie daher für die jeweilige Variante die bereitgestellten Tabellen aus.

➔ Lösen Sie diese Aufgabe auf der nächsten Seite.

1. Art der Variablenübergabe: Call by Referenz

```
01 #include <iostream>
02
03 int i;
04 int b[2];
05
06 void p(int &x)
07 {
08                  // -3-
09     i = i + 1;
10     x = x + 2;
11                  // -4-
12 }
13
14 int main()
15 {
16     b[0] = 10;
17     b[1] = 20;
18     i = 0;
19                 // -1-
20     p (b[i]);
21                 // -2-
22 }
```

Hauptprogramm

Stelle	b[0]	b[1]	i
-1-	10	20	0
-2-	12	20	1

Unterprogramm

Stelle	x
-3-	10
-4-	12

2. Art der Variablenübergabe: Call by Zeigervariable

```
01 #include <iostream>
02
03 int i;
04 int b[2];
05
06 void p(int *x)
07 {
08                  // -3-
09     i = i + 1;
10     *x = *x + 2;
11                  // -4-
12 }
13
14 int main()
15 {
16     b[0] = 10;
17     b[1] = 20;
18     i = 0;
19                 // -1-
20     p (&b[i]);
21                 // -2-
22 }
```

Inhalt ist eine beliebige Adresse.
Muss nur bei 3 u. 4 gleich sein.

Hauptprogramm

Stelle	b[0]	b[1]	i
-1-	10	20	0
-2-	12	20	1

Unterprogramm

Stelle	x
-3-	Adresse
-4-	Adresse

Aufgabe 5 Kontrollstrukturen und Sortieralgorithmus

Gegeben ist eine einfache verkettete Liste *VKL* aus mehreren Elementen vom Typ "Graphelement". Zur verketteten Liste gibt es einen Zeiger auf das erste Element (*VKL_start*) und einen Zeiger auf das letzte Element (*VKL_ende*). Zusätzlich zeigt die Variable *next* von jedem Graphelement der verketteten Liste auf das nächste Element in der Liste. Beim letzten Element zeigt die Variable *next* auf NULL. Außerdem hat jedes Element vom Typ "Graphelement" einen Zeiger auf eines der Gatter-Elemente, das die charakteristischen Informationen zum jeweiligen Gattertyp beinhalten (wie z.B. Grundlaufzeit, Lastfaktor, …). Ferner hat jedes Graphelement seine Verbindungen zu anderen Graphelementen des Graphen im Array *graph[]* gespeichert (die Anzahl der Ziele ist in der Variablen *anzahl_graph_ziele* gespeichert und kann maximal den Wert 5 haben). Des Weiteren gibt es eine weitere globale Variable *n*, die die Anzahl der Graphelemente der verketten Liste speichert und drei weitere Konstanten *Kt*, *Kv* und *Kp*, die die äußere Faktoren enthalten.

```
struct Gatter {
     string typ;
     double grundlaufzeit;
     short lastfaktor;
     short lastkapazitaet;
};

struct Graphelement
{
   string name;
   Gatter* typ;
   Graphelement* next;
   Graphelement* graph[5];
   int anzahl_graph_ziele;
   Graphelement* vater;
   bool isEnde; //Zeigt, ob das Gatterelement am Ende des Pfades ist
   double laufzeit;

};

Graphelement* VKL_start = NULL;
Graphelement* VKL_ende = NULL;

const double Kv =1.09, Kt =0.94, Kp =1;

int n;
```

Hinweis: Es soll angenommen werden, dass die verkettete Liste schon aufgebaut ist, die Zeiger *VKL_start* und *VKL_ende* auf das jeweilige Graphelement zeigen und die Datenelemente der einzelnen Graphelemente bis auf die Komponente *laufzeit* komplett gefüllt sind.

A) Schreiben Sie eine for-Schleife, die die Anzahl der Elemente der verketteten Liste ermittelt und in die Variable *n* speichert. Evtl. verwendete Variablen müssen definiert / deklariert werden.

```
n = 0;

for(Graphelement* ptr_Graph = VKL_start; ptr_Graph != NULL;
    ptr_Graph = ptr_Graph->next)
       n++;

 // oder

for(Graphelement* ptr_Graph = VKL_start; ptr_Graph != NULL;
    n++)
   ptr_Graph = ptr_Graph->next;
```

B) Die Laufzeit eines Graphelements ist nach der Formel 5.1 von der Last am Ausgang und den äußeren Faktoren abhängig. Nun soll für jedes Graphelement diese Laufzeit berechnet werden.

$$t_{pd,actual} = t_{pd0} + K_L \cdot C_{last} \cdot K_T \cdot K_V \cdot K_P \qquad (5.1)$$

wobei:

$t_{pd,actual}$: last- und faktorenabhängige Laufzeit eines Graphelements in *ps*
t_{pd0}: Grundlaufzeit in *ps*
K_L: Lastfaktor in *fs/fF*
C_{last}: Lastkapazität in *fF*; ist gleich der Summe der Lastkapazitäten der am Ausgang angeschlosssenen Graphelemente
K_T, K_V, K_P: Temperatur, Versorgungsspannung und Prozesstoleranz Faktoren, die jeweils in den Konstanten *Kt*, *Kv* und *Kp* gespeichert sind

Ergänzen Sie hierzu die folgende Funktion:

```
void Laufzeit_berechnen(Graphelement* ptr_Graph)
{
 int last=0;
 for(int i = 0; i < ptr_Graph->anzahl_graph_ziele; i++)
 {
       last+=ptr_Graph->graph[i]->typ->lastkapazitaet;
 }
 ptr_Graph->laufzeit=(ptr_Graph->typ->grundlaufzeit * 1000 + \
                          ptr_Graph->typ->lastfaktor * last) * \
                          tmp_Faktor * spg_Faktor * prz_Faktor / 1000;









}
```

C) Erzeugen Sie nun ein dynamisches Array mit dem Namen *Arr_ptr*, das aus *n* Elementen vom Typ „Zeiger auf Graphelement“ besteht.

```
Graphelement** Arr_ptr = NULL;

Arr_ptr = new Graphelement* [n];
```

D) Im nächsten Schritt soll jedes Feld des Arrays *Arr_ptr* an der i-ten Stelle ($0 \leq i \leq n-1$) des Arrays auf das i-te Graphelement in der verketteten Liste zeigen. Evtl. verwendete Variablen müssen definiert / deklariert werden.

```
int i=0;

for(Graphelement* ptr_Graph = VKL_start; ptr_Graph != NULL;
    ptr_Graph = ptr_Graph->next)
{

    Arr_ptr[i] = ptr_Graph;
    i++;

}
```

E) Nun soll der Insertionsort-Algorithmus angewendet werden, um die Graphelemente nach deren Laufzeit ansteigend zu sortieren. Hierzu soll das Sortieren anhand der „Zeiger auf Graphelement“ des Arrays *Arr_ptr* durchgeführt werden.

a) Im Folgenden ist der Pseudocode des Insertionsort-Algorithmus gegeben:

```
INSERTIONSORT(A)
1 for j = 1 to Länge[A]-1
2    do Schlüssel = A[j]
         //dann Füge A[j] in die sortierte Folge A[1 .. j − 1] ein.
3        i = j − 1
4        while i ≥ 0 und A[i] > Schlüssel
5            do A[i + 1] = A[i]
6                i = i − 1
7        A[i + 1] = Schlüssel
```

Setzen Sie die Funktion *INSERTIONSORT* in C++ Code um. Passen Sie dabei den Pseudocode der Problemstellung an (in Bezug auf den Vergleich der Laufzeiten).

Nachstehend ist der Prototyp dieser Funktion vorgegeben. Evtl. verwendete Variablen müssen definiert / deklariert werden.

```
void INSERTIONSORT (Graphelement* A[])
{

   Graphelement* Sch; //Schlüssel

   signed int i;

   for (int j=1; j < n; j++)

   {
       Sch= A[j];
       i= j-1;
       while ((i>=0)&&( A[i]->laufzeit > Sch->laufzeit))
       {
          A[i+1]= A[i];
          i=i-1;
       }
       A[i+1]= Sch;
   }









}
```

Aufgabe 6 Studentenverwaltung

Das Institut für Technik der Informationsverarbeitung möchte für die Klausur Informationstechnik ein Tool in C++ entwickeln, um die Studenten und ihre Note zu verwalten. Sie haben die Aufgabe, dieses Tool zu programmieren.

Die Struktur, in der die Daten der Studenten gespeichert sind, sieht folgendermaßen aus:

```
struct student {
  string name;
  short matrikel;
  float note;
  student* next;
};
```

Da die Anzahl der Teilnehmer an der Klausur nicht vorhergesehen werden kann, sollen die Daten in einer speziellen verketteten Liste gespeichert werden. Die verkettete Liste soll wie in Abbildung 6.1 aufgebaut sein. Bitte beachten Sie dabei, dass das letzte Element wieder auf das erste Element der verketteten Liste zeigt und dass kein Zeiger für das Ende der verketteten Liste vorhanden ist.

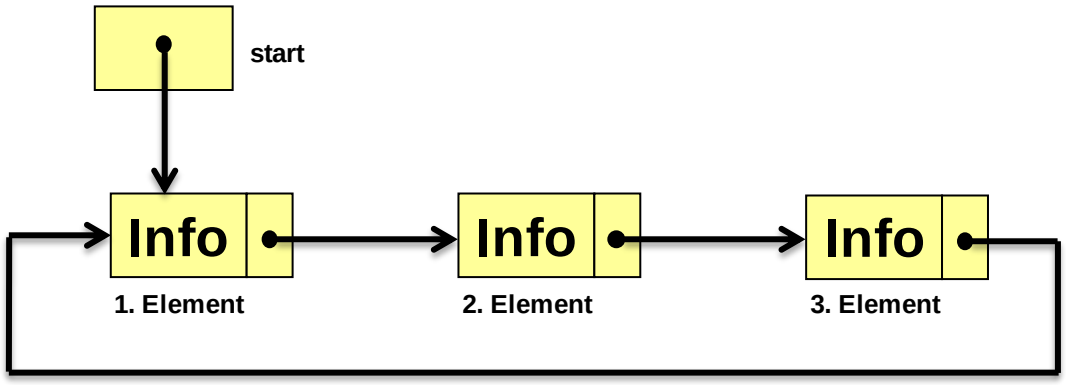


Abbildung 6.1 prinzipieller Aufbau der verketteten Liste

Beim Starten des Programms sollen die Daten aus einer Datei eingelesen werden und die verkettete Liste entsprechend aufgebaut werden. Die folgende Abbildung zeigt einen Ausschnitt aus der Datei.

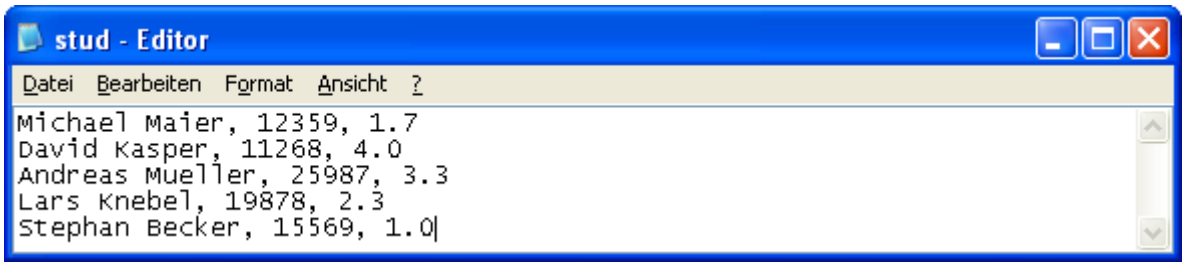


Abbildung 6.2 prinzipieller Aufbau der Datei

A) Geben Sie den Programmcode an (nur eine Zeile), um die oben stehende Datei *stud.txt* zum Lesen zu öffnen. Und geben Sie zwei weitere Zeilen Programmcode an, mit welchen eine Fehlermeldung ausgegeben wird, wenn das Öffnen der Datei nicht erfolgreich war.

```
ifstream file( "stud.txt" );
if ( !file )
  cout << "Fehler beim Oeffnen der Datei" << endl;
```

B) Wie würde eine Schleife aussehen, mit der die Datei Zeilenweise bis zum Dateiende ausgelesen wird. Bitte geben Sie nur den Schleifenkopf an. Eventuell verwendete Variablen müssen vorher deklariert werden.

```
string zeile;
while (getline( file, zeile ))
 { ... }
```

C) Es sei eine Zeile aus der Datei ausgelesen und der Funktion *Zeile_auswerten()* übergeben. Ergänzen Sie diese Funktion. Diese soll mit Hilfe von Befehlen zur Stringmanipulation den String in der Übergabevariablen *zeile* in seine verschiedenen Bestandteile (Name, Matrikelnummer, Note – ebenfalls Varibalentyp String) zerlegen und den entsprechenden Variablen zuweisen. Es kann davon ausgegangen werden, dass alle Zeilen das gleiche Format haben und immer alle Werte vorhanden sind, wie in Abbildung 6.2 gezeigt. Die String-Variablen werden anschließend, wenn nötig, umgewandelt und dann der Funktion *Element_hinzufuegen()* übergeben – siehe Aufgabenteil D).

```
void zeile_auswerten(string zeile)
{
  string name;
  string s_matrikel;
  string s_note;

  int pos1 = zeile.find(",", 0);;
  int pos2 = zeile.find(",", pos1+1);
  name = zeile.substr(0, pos1);
  s_matrikel = zeile.substr(pos1+1, pos2-pos1-1);
  s_note = zeile.substr(pos2+1, zeile.length() - pos2);
```

```
  short matrikel = matrikel_umwandeln(s_matrikel);
  float note = note_umwandeln(s_note);

  Element_hinzufuegen(name, matrikel, note);
}
```

D) Im nächsten Schritt soll die Funktion *Element_hinzufuegen()* implementiert werden, welche ein neues Element in der verketteten Liste erzeugt und die Daten darin ablegt. Die Funktion bekommt den Namen, die Matrikelnummer und die Note übergeben. Der Startzeiger *start* der verketteten Liste ist eine globale Variable, welche zu Beginn mit NULL initialisiert ist. Die Funktion soll ein neues Element für die verkettete Liste erzeugen, dieses mit Daten füllen und zur verketteten Liste hinzufügen.
Hinweis: Das Element muss **nicht** am Ende der verketteten Liste hinzugefügt werden.

```
void Element_hinzufuegen(string name, short matrikel, float note)
{
  student* neu = new student;

  neu->name = name;
  neu->matrikel = matrikel;
  neu->note = note;

  if (start == NULL) {
       start = neu;
       neu->next = start;
  }
  else {
       neu->next = start->next;
       start->next = neu;
  }









}
```

E) Im Programm soll zusätzlich die Funktion *Element_suchen()* implementiert werden. Diese erhält als Übergabeparameter die Matrikelnummer des Studenten und hat als Rückgabewert einen Zeiger auf *Student*. Dieser zeigt auf das gesuchte Element, falls es in der Liste vorhanden ist, ansonsten ist der Inhalt NULL. Schreiben Sie zu dieser Funktion den entsprechenden Pseudocode.

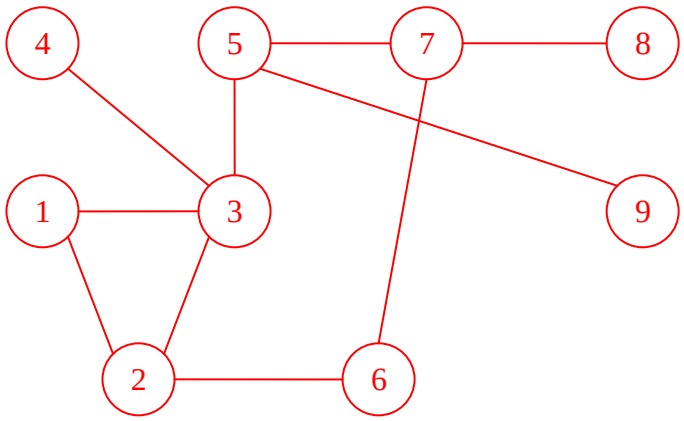
```
Zeiger auf Student ptr anlegen
Zeiger auf Student: Ergebnis anlegen & NULL initialisieren

Wenn start ist nicht NULL // Wenn Liste ist nicht leer
dann ptr = start
     Schleifen_start
       Wenn ptr→matrikel gleich gesuchter Matrikelnummer
       dann Ergebnis = ptr
            Schleife verlassen
       Ende_Wenn
       ptr = ptr → next // Zum nächsten Element wechseln
     Schleife_ende solange ptr ist nicht start
        //solange aktuelles Element ist nicht das Startelement
Ende_Wenn
Ergebnis zurückgeben
```

Aufgabe 7 Graphentheorie

A) Folgende Adjazenzmatrix beschreibt einen Graphen. Zeichnen Sie diesen.

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
1	0	1	1	0	0	0	0	0	0
2	1	0	1	0	0	1	0	0	0
3	1	1	0	1	1	0	0	0	0
4	0	0	1	0	0	0	0	0	0
5	0	0	1	0	0	0	1	0	1
6	0	1	0	0	0	0	1	0	0
7	0	0	0	0	1	1	0	1	0
8	0	0	0	0	0	0	1	0	0
9	0	0	0	0	1	0	0	0	0

Lösung:



B) *Kreuzen Sie die richtige Antwort an. Der Graph aus A) ist:

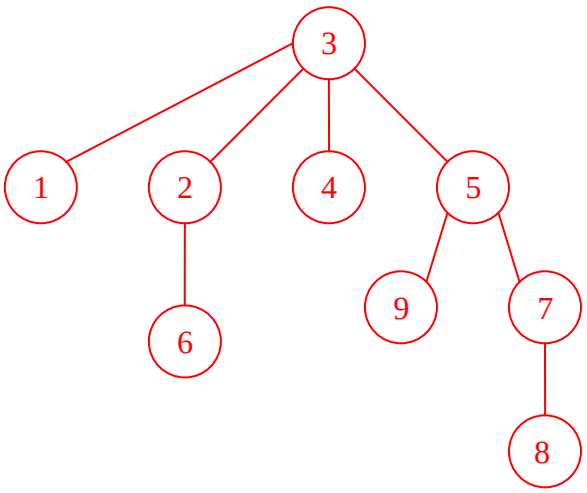
	ja	nein
zusammenhängend	x	
gerichtet		x

C) *Wenden Sie nun auf den Graphen von A) eine Breitensuche mit Hilfe des im Folgenden vorgegebenen Pseudocodes an. Der Startknoten ist Knoten 3. Füllen Sie hierzu die vorgegebene Tabelle aus. Bei mehreren Nachfolgern, werden die Knoten in ihrer numerischen Reihenfolge gefunden. Im Pseudocode steht *Q* für eine Warteschlage.

```
   BFS(G,s)
01.    for alle Knoten u ∈ V\{s}
02.      do farbe[u]=weiss;
03.         d[u]= ∞;
04.         π[u]=NIL;
05.    farbe[s]=grau; d[s]=0; π[s]=NIL;
06.    Q=∅;
07.    ENQUEUE(Q,s);    //s in die Warteschlange hinzufügen
08.    while (Q≠∅)
09.      do u=DEQUEUE(Q) //erstes Element aus der Warteschlage nehmen
10.         for alle v ∈ Adj[u]
11.           do if farbe[v]==weiss
12.              then farbe[v]=grau; d[v]=d[u]+1; π[v]=u;
13.                   ENQUEUE(Q,v);
14.         farbe[u]=schwarz;
15.    end
```

Bearbeiteter Knoten	Inhalt der Warteschlange
-	3
3	1,2,4,5
1	2,4,5
2	4,5,6
4	5,6
5	6,7,9
6	7,9
7	9,8
9	8
8	-

D) *Zeichnen Sie den sich ergebenden Graphen aus Aufgabe C).



E) *Wie nennt man den sich ergebenden Graphentyp aus Aufgabenteil D)?

ein Baum

Aufgabe 8 Algorithmus Analyse

Gegeben ist der folgende Algorithmus für Merge-Sort (Sortieren durch Mischen) in Pseudocode. Dabei ist der Übergabeparameter A das zu sortierende Array, p der Start-Index des zu sortierenden Bereiches des Arrays und r der Index des Endelements des zu sortierenden Bereiches des Arrays.

Hinweise:

I. Die Funktion Merge-Sort wird aufgerufen für Eingabearrays, deren Länge eine Potenz von 2 ist.

II. Das erste Feld jedes benutzten Arrays hat den INDEX 1.

```
Merge-Sort (A, p, r)
1.  if (p < r)
2.     then q ← (p + r -1) /2
3.          Merge-Sort (A, p, q)
4.          Merge-Sort (A, q + 1, r)
5.          Merge (A, p, q, r)
```

```
Merge (A, p, q, r)
01.  t ← r – p + 1
02.  erzeuge T [1..t] und fülle es mit 0   // (2t+1) Rechenschritte für diese Zeile!
03.  i ← p
04.  j ← q + 1
05.  k ← 1
06.  solange ((i ≤ q) und (j ≤ r))
07.       do if (A[i] ≤ A[j])
08.            then  T[k] ← A[i]
09.                  i ←i + 1
10.            else  T[k] ← A[j]
11.                  j ←j + 1
12.          k ← k + 1
13.  solange (i ≤ q)
14.       do T[k] ← A[i]
15.          i ← i + 1
16.          k ← k + 1
17.  solange (j ≤ r)
18.       do T[k] ← A[j]
19.          j ← j + 1
20.          k ← k + 1
21.  antworte T[ ]                  // (2t+1) Rechenschritte für diese Zeile!
```

A) Gegeben sei folgendes Array:

A:

4	7	3	2

Nach dem Aufruf "**Merge-Sort** (A, 1, Länge(A))", befindet man sich im letzten Ausführungsschritt des Sortieralgorithmus, wo nur noch die Funktion "**Merge** (A, 1, 2, 4)" ausgeführt werden muss.

Füllen Sie folgende Tabellen aus.

- Array A unmittelbar vor der Durchführung der Funktion "**Merge** (A, 1, 2, 4)":

A :

4	7	2	3

- Array T während der Durchführung der Funktion "**Merge** (A, 1, 2, 4)":

Hinweis: Verwenden Sie für jede Veränderung der Variablen i, j oder k eine neue Zeile in der Tabelle.

i	j	k	T [Index] Index=1	Index=2	Index=3	Index=4
			0	0	0	0
1	3	1	2	0	0	0
1	4	2	2	3	0	0
1	5	3	2	3	4	0
2	5	4	2	3	4	7
3	5	5	2	3	4	7

B) Führen Sie die Laufzeit-Analyse durch für die Funktion **Merge**, wenn diese wie folgt aufgerufen wird: "**Merge** (A, 1, n/2, n)" (dies ist der letzte Funktionsaufruf innerhalb der Funktion **Merge-Sort** nach dem Aufruf **Merge-Sort** (A, 1, n), bevor das Array A der Länge n vollständig sortiert ist).
Zu beachten ist, dass n eine Potenz von 2 ist.
Außerdem wird angenommen, dass der kleinste Wert in der linken Hälfte vom Array A (Index 1 bis n/2) größer ist, als der größte Wert in der rechten Hälfte vom Array A (Index (n/2)+1 bis n) ist. → Das heißt A[1] > A[n].

➔Füllen Sie zum Lösen der Aufgabe die Tabelle auf der nächsten Seite aus.

Zeile	Kostenkoeffizient	Zeit
1	c_1	1
2	c_2	2n +1
3	c_3	1
4	c_4	1
5	c_5	1
6	c_6	n/2 + 1
7	c_7	n/2
8	c_8	0
9	c_9	0
10	c_{10}	n/2
11	c_{11}	n/2
12	c_{12}	n/2
13	c_{13}	n/2 + 1
14	c_{14}	n/2
15	c_{15}	n/2
16	c_{16}	n/2
17	c_{17}	1
18	c_{18}	0
19	c_{19}	0
20	c_{20}	0
21	c_{21}	2n + 1

Bestimmen Sie aus der ausgefüllten Tabelle eine obere asymptotische Schranke für die Laufzeit T(n) der Funktion **Merge**:

$$T(n) = O(\_\_n\_)$$